Suplemento mensual dedicado a la Imagen Sintética

Número20

RmEaNnDíEaR

Taller Virtual
La potencia de las sentencias de programación

Cómo...
Crear escenas con radiosidad

Preview
Presentamos la nueva versión de POV

Foro del Lector
Reunión de mechs

pcmanía

José Manuel Muñoz

Sentencias de programación (parte I)

Comenzamos este nuevo número de Rendermanía con las llamadas sentencias de programación de «POV». Actuaremos así por tres razones: la primera porque existen lectores que confiesan no comprender su funcionamiento, la segunda porque el dominio de estas sentencias es necesario para poder aprovechar al máximo las posibilidades de construcción y animación que ofrece «POV» y la tercera, porque ya empieza a hablarse de los futuros cambios que en este apartado van a aparecer con la próxima versión de «POV».

Antes de empezar a estudiar el proceso de creación de animaciones con «POV» conviene repasar el funcionamiento de las sentencias "de programación" de «POV». Además, aunque ya hemos hablado de #if, #while y de otras sentencias de este tipo en números anteriores, el caso es que sigue habiendo un gran número de usuarios que –aun siendo talentosos pov-artistas– no

comprenden su funcionamiento. Y, sin embargo, la importancia de estas sentencias es enorme.

Con ellas se pueden levantar ciudades, disponer ejércitos, crear estructuras complejas para naves espaciales, etc. De hecho, les hemos concedido una importancia mucho mayor de la que otorga a las sentencias que se utilizan en «POV» para crear objetos. ¿Por qué? Porque en «POV» no es indispensable que creemos los objetos que van a aparecer en la escena con su lenguaje escénico. Se pueden emplear programas como «3D Studio», «Imagine», «Spatch» o «Rhino» para crear los objetos y luego exportarlos a «POV» y renderizar las escenas desde este último. Como recordaréis, «Spatch» y «Rhinoceros» pueden exportar directamente sus modelos a «POV», mientras que para hacer lo propio desde «3D Studio» e «Imagine» deberemos emplear utilidades –ya publicadas aquí en números anteriores– como 3ds2pov y 3dto3d, respectivamente.

Este ha sido, además, el método de trabajo empleado en los últimos números de Rendermanía y puede resumirse en los siguientes pasos:

1) Uso de un modelador para crear los modelos básicos de la escena.

2) Exportación de estos a «POV». En este paso el programa empleado creará uno o más ficheros ASCII para guardar los modelos en un formato inteligible para «POV», normalmente utilizando mesh, triangle, smooth_triangle y otras sentencias del lenguaje escénico de «POV».

3) Empleo de un editor de textos para crear los ficheros de escena. En ellos se escribirán declaraciones utilizando el lenguaje de «POV» para situar la cámara, las luces y los objetos importados (que se hallan almacenados en otros ficheros a los que se hace referencia con la sentencia #include). También se crearán desde aquí las texturas procedurales necesarias.

4) Pruebas y pruebas de render.

Como veis, estos pasos sólo requieren comprender una parte de lo que el lenguaje escénico de «POV» puede hacer. No es preciso conocer las primitivas de «POV» para crear objetos, pero sí es necesario saber cómo funcionan las transformaciones espaciales y cómo se sitúan la cámara, los objetos y las luces. También es conveniente, aunque no imprescindible, comprender cómo funcionan las texturas procedurales. (Nota: todas estas cosas han sido explicadas en los números 16, 17, 18 y 19 de Rendermanía y recomendamos a los usuarios novatos de «POV» leer dichos números si quieren entender algo).

Esta filosofía nos permitirá disfrutar de las ventajas visuales que conlleva el trabajo con los modeladores interactivos que utilizan ventanas al tiempo que podremos gozar, también, de las ventajas que «POV» posee con respecto a la mayoría de dichos programas. Aquí nos estamos refiriendo a las sentencias con que cuenta «POV» para crear magníficas texturas procedurales, a la posibilidad de situar miles de objetos poligonales sin excesivo gasto de memoria –sentencia mesh–, a la calidad de su render, etc.

Por otra parte, las sentencias de programación no son difíciles de utilizar, aunque pueden suponer un pequeño esfuerzo para quienes nunca hayan trabajado con lenguajes de programación.

Aquí se ha usado el #while para crear los peldaños de la escalera.

Bucles con #while

Indiscutiblemente la colocación de objetos puede ser –con independencia del programa que se esté empleando– una de las tareas más molestas en la construcción de ciertas escenas. Crear imágenes como las aparecidas en las portadas de los números 14 y 17 podría ser una auténtica pesadilla si no dispusiéramos de alguna ayuda que nos evitara colocar manualmente cada modelo dentro de la escena. Probablemente, pensaréis que raramente es preciso

Figura 1: "ejemplos sencillos de uso de #while"

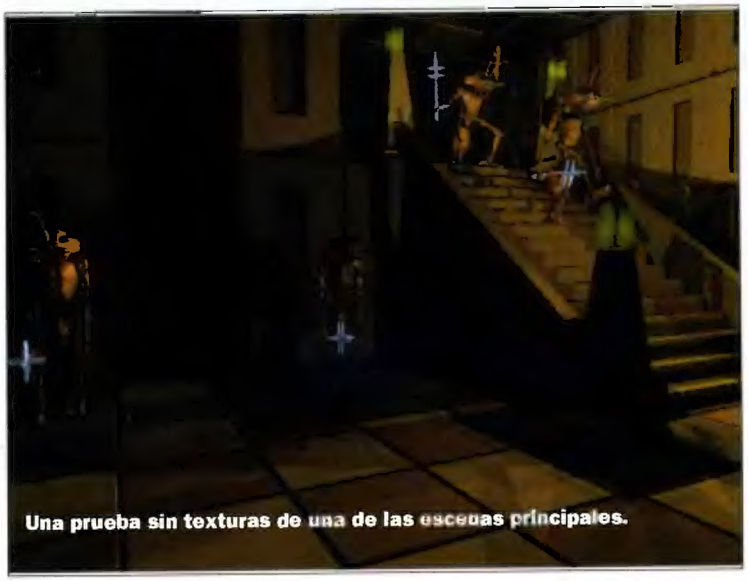
```
//ejemplo A
#declare nz=0
#while(nz<40)
        object{Bcolum150_300s1 translate<0,0,nz*150>}
        object{Bcolum150_300s1 translate<450,0,nz*150>}
        #declare nz=nz+1
#end


//Ejemplo B: este trozo de código hace lo mismo que el del ejemplo anterior,
pero es mas útil.
#declare nz=0
#declare galeria1_3x3y=union{
    #while(nz<40)
        object{Bcolum150_300s1 translate<0,0,nz*150>}
        object{Bcolum150_300s1 translate<450.0.nz*150>}
        #declare nz=nz+1
    #end
}
object{galeria1_3x3y}
```

crear escenas con tantos elementos repetidos, pero lo cierto es que resulta posible imaginar muchos ejemplos de colocación tediosa de objetos. Por ejemplo, los tacos de las llantas de una bicicleta (o los radios de sus ruedas), las ventanas de un edificio, las columnas de un templo, los peldaños de una escalera (del tipo que sea), un castillo de naipes, los libros de una estantería, etc. Programas como «Rhino» y «MAX» tienen opciones para colocar arrays de objetos en formaciones cuadradas, circulares o de manera que sigan un patch, pero hay algunas limitaciones como la posibilidad de crear variaciones particulares en cada objeto del grupo. (Nota: esto no es del todo exacto. Tenemos entendido que existe un Ipas para «3D Studio» que permite hacer lo mismo y se dice que el lenguaje escénico de la versión 2.5 de «MAX» permite hacer cosas increíbles en este y otros apartados).

Pues bien, en «POV» podemos crear cualquier escena de este tipo si comprendemos bien cómo debe emplearse #while. Esta sentencia funciona prácticamente igual que sus equivalentes en lenguajes de programación como C, aunque la sintaxis es algo diferente. Para emplear #while tendremos que establecer una condición que definiremos dentro de los paréntesis que siguen a la sentencia. Luego escribiremos una lista de instrucciones comprendidas entre las sentencias #while y #end. A esta lista de sentencias la llamaremos el cuerpo del bucle y todas ellas se repetirán tantas veces como se cumpla la condición escrita en la sentencia #while.

Veamos cómo funciona esto con algo más de detalle. Examinad el código de la Figura 1. En ella tenemos dos ejemplos de bucles simples. En el ejemplo A la primera sentencia es una declaración por la cual se otorga al identificador o variable nz el valor 0. (Nota: si no recordáis cómo funcionaba #declare, echad un vistazo al número 17). La siguiente sentencia es la instrucción #while y su condición puede leerse como "mientras nz sea menor que 40". Las tres sentencias siguientes componen el cuerpo del bucle y se ejecutarán una y otra vez hasta que nz sea igual al valor 40. en cuyo momento «POV» continuará procesando las sentencias escritas después del #end. Veamos esto paso a paso: la primera sentencia del cuerpo del bucle crea el objeto "Bcolum150_300s1" (que supuestamente ha sido definido en líneas anteriores). Este objeto se creará en la posición <0,0,0> porque –por ahora– nz vale cero. La siguiente sentencia creará otro objeto del mismo tipo en la posición <450,0,0> (porque nz sigue valiendo cero). Después nz sumará 1 a su valor anterior y al llegar el parser de «POV» (ver número 16) a la sentencia #end, esta le hará cerrar el bucle y retornar al #while para volver a considerar la condición. Entonces se volverán a procesar las dos primeras



Una prueba sin texturas de una de las escenas principales.

sentencias del cuerpo del bucle y se crearán dos nuevos objetos, aunque esta vez, como nz vale 1, los objetos se crearán en las posiciones <0,0,150> y <450,0,150>. (Tomad nota en cada ejemplo de la forma en que se emplean las transformaciones para situar espacialmente cada objeto creado por #while). Al ejecutarse la tercera sentencia, nz pasará a valer 2 y al llegar al #end el parser volverá a estudiar la condición de la sentencia #while. Esto sucederá una y otra vez hasta que nz valga 40 en cuyo momento la condición dejará de cumplirse y el parser abandonará el bucle para comenzar a estudiar las sentencias que sigan al #end.

En este ejemplo, el bucle nos servirá para crear 80 objetos "Bcolum150_300s1". Naturalmente, también podríamos haber escrito manualmente las declaraciones de los 80 objetos, pero esto, sabiendo utilizar #while, sería una estupidez. Además, en caso de haber cometido un error de posicionamiento o de otro tipo, será mucho más fácil corregir las pocas sentencias que forman el bucle que todas las líneas erróneas escritas manualmente. También, de igual manera, nos resultaría sencillo alterar el bucle para crear cientos o miles de objetos en lugar de sólo 80, ya que bastaría con cambiar la condición del bucle.

Por otro lado, hemos de tener cuidado con la condición escrita dentro del #while y con las variables implicadas. Supongamos que hubiésemos olvidado incluir la sentencia #declare nz=nz+1. En este caso nz seguiría valiendo siempre cero, sin importar cuántas pasadas se hiciesen, con lo cual el compilador de «POV» quedaría encerrado dentro de un bucle sin fin. ¿Sin fin? Bueno la verdad es que no. Hay que recordar que en este ejemplo estamos creando dos objetos en cada nueva pasada del bucle. Estos objetos ocupan una cantidad determinada de memoria (por pequeña que sea) y esto quiere decir que tarde o temprano «POV» nos soltaría un mensaje de memoria insuficiente y detendría la compilación sin haber renderizado ni una sola línea. (Por otro lado, como en el caso comentado nz siempre valdrá cero, lo que sucederá es que se irán creando pares de objetos, siempre en las mismas posiciones iniciales. Por tanto, si pudiéramos interrumpir el parsing y ver los objetos creados, sólo veríamos 2, aunque realmente se habrían creado cientos o quizá miles de ellos en la mismas posiciones).

Prueba para portada.

También podríamos cometer errores de otro tipo. Las sentencias de programación como #while son sencillas de comprender pero pueden provocar verdaderos desastres por culpa de pequeños despistes. Si hubiésemos olvidado escribir #declare nz=0 antes del bucle y hubiésemos empleado a esta variable en bucles o cálculos anteriores, podría suceder muy bien que nz llegase al bucle con un valor superior a 40, con lo cual –como la condición se estudia antes de realizar una sola pasada– no se realizaría pasada alguna ni se crearía ningún objeto.

Ahora consideremos otra cuestión. Con el ejemplo comentado crearemos un grupo determinado de objetos en la escena. Ahora bien, ¿qué sucede si deseamos crear otro grupo de objetos igual en otra posición? Supongamos que hemos utilizado un bucle para crear los 100 peldaños de una escalera y que hemos de crear varias escaleras de este tipo en un edificio. Aunque nos hayamos ahorrado la escritura manual de los 100 peldaños resultaría tedioso escribir el mismo bucle #while una vez para cada escalera a crear (alterando únicamente los valores de posición de los objetos en cada caso). En lugar de esto sería mucho más práctico hacer algo como lo que podéis ver en el ejemplo B de la Figura 1. En dicho ejemplo se ha declarado una unión de objetos (ver número 17) y se ha escrito el mismo bucle #while de antes dentro de di-

cha unión. De esta forma, podremos crear al grupo de objetos de la unión tantas veces como queramos en el espacio de «POV». Para ello simplemente invocaremos a la unión dentro de una sentencia object, tal como puede verse en la última línea del ejemplo. Y para situar a cada copia del grupo en una posición diferente, únicamente tendremos que escribir una orden translate apropiada dentro de cada sentencia object. Quizá convenga recordar que con #declare los objetos sólo se están creando en la memoria de «POV». Únicamente aparecerán en el pov-espacio cuando los objetos declarados sean referenciados con object. También conviene que recordéis que los desplazamientos provocados por translate son sumas que se añaden a las posiciones anteriores de los objetos.

Por último, consideremos otro caso más. Siguiendo con el ejemplo anterior supongamos que en nuestro edificio hay muchas escaleras del mismo tipo pero que puede haber un número diferente de peldaños en cada escalera. En este caso quizá podríais sentir la tentación de escribir algo parecido al trozo de código de la Figura 2.

En este ejemplo hemos definido al objeto peldanno (para «POV» el carácter de la eñe es ilegal) como una unión de 3 objetos; cada peldaño está compuesto por 3 piedras. Posteriormente, este objeto es referenciado desde dentro de la siguiente unión, con la que construiremos la escalera, lo que es totalmente correcto. En el ejemplo intentamos crear dos escaleras, una de 20 peldaños y otra de 40. Para ello asignamos un valor a la variable que establece el límite del contador de peldaños y luego invocamos a tramo_escalera. Sin embargo, esto no funcionará.

Si la variable nmaxpel no ha sido declarada antes de que el parser de «POV» llegue al #while, entonces aparecerá un error diciendo que esa variable no existe, aunque se la declare unas líneas más abajo. Y si damos un valor de, por ejemplo, 30 a nmaxpel, entonces las escaleras serán siempre de 30 peldaños, independientemente del valor que asignemos más adelante a nmaxpel antes de cada invocación a tramo_escalera. Esto sucede porque el parser de «POV» sólo procesa una vez la declaración de tramo_escalera y por ello cada invocación a esta unión siempre creará el mismo número de objetos.

Figura 2: "este ejemplo de #while no funcionará correctamente"

```
//ESCALERAS
#declare peldanno=union{
    object{Bsuelo100_100d1}
    object{Bsuelo100_100d2 translate<100,0,0>}
    object{Bsuelo100_100d1 translate<200,0,0>}
}
#declare tramo_escalera=union{
    #declare npel=0
    #while (npel<nmaxpel)
        object{peldanno translate<0,-20*npel,-28*npel>}
        #declare npel=npel+1
    #end
}
#declare nmaxpel=20
object{tramo_escalera translate<400,500,1000>}

#declare nmaxpel=40
object{tramo_escalera translate<1400,500,2000>}
```

Otra prueba con iluminación de las escaleras.

Sin embargo, existe una solución: si sacamos del fichero toda la declaración de tramo_escalera y la almacenamos en otro archivo que sea leído y procesado nuevamente, cada vez que queramos crear la escalera, la cosa funcionará. Con este sistema tendríamos que crear cada escalera escribiendo algo similar a lo que puede verse en la Figura 3.

En esta figura hay trozos de código de dos archivos diferentes. Por un lado tenemos el bucle #while que creará la escalera y por otro las sentencias object dentro de las que hemos escrito los includes. Cuando el parser de «POV» halle el primer #include (número 17 de Rendermanía) se cargará y procesará el archivo escalera.inc donde hemos escrito el bucle. Como nmaxpel ha sido declarada antes del #while guardado en el archivo que se va a leer, no se producirá ningún error y, en el primer caso, se creará una escalera de 20 peldaños. Luego se asignará a nmaxpel el valor 40 y volverá a leerse y a procesarse al fichero escalera.inc, con lo que se creará una nueva escalera de 40 peldaños. Esto lo conocemos con el nombre de pov-macro (aunque otros la llaman una pov-función). Realmente, y tal como lo ve «POV», esto es equivalente a escribir dos veces el bucle #while pero para nosotros la cosa funcionará

figura 3: "este ejemplo de #while sí funcionará correctamente"

```
//este trozo de código es el archivo escalera.inc
union{
    #declare npel=0
    #while (npel<nmaxpel)
        object{peldanno translate<0,-20*npel,-28*npel>}
        #declare npel=npel+1
    #end
}

//el siguiente código está incluido dentro del archivo escénico con que se invoca a POV.
El bucle #while //está en otro archivo llamado escalera.inc
(que sólo almacena ese trozo de código)
#declare nmaxpel=20
object{#include"escalera.inc" translate<400,500,1000>)

#declare nmaxpel=40
object{#include"escalera.inc" translate<1400,500,2000>)
```

tal como lo hace una función de C.

Si queremos crear 40 escaleras en distintas posiciones bastará con invocar 40 veces a la pov-macro, aunque podría ocurrir que pudiéramos incluir las invocaciones dentro de un nuevo bucle #while, lo que nos permitiría ahorrar aún más líneas de código.

Bucles anidados

Por último vamos a hablar de la anidación de bucles. En las escenas del presente número hemos utilizado un conjunto de objetos muy simple –bloques de piedra de pared y suelo, peldaños, etc.– para construir la arquitectura del Vormiguero. El resultado es gráficamente bastante simple pero resulta muy adecuado para experimentar con el funcionamiento de las sentencias de programación, ya que todo –las paredes, los suelos, las escaleras, los puentes, etc.– ha sido creado empleando bucles para situar las docenas o cientos de objetos simples que componen cada estructura. Además, los interiores resultaban adecuados para hacer pruebas de radiosidad.

El suelo de una sala, por ejemplo, puede servirnos para explicar el asunto de los bucles anidados. En estas escenas tanto los suelos como los techos están formados por una matriz de bloques simples. En los suelos, uno de los objetos más utilizados es un bloque de 2 metros de ancho (eje X) por 2 de profundidad (eje Z) y medio metro de altura (eje Y). Así, el suelo de una sala de 40 metros de ancho por 20 de profundidad estará formada por 20*10 bloques de este tipo. Desde luego escribir a mano las 200 sentencias object para situar cada bloque de suelo es algo impensable. ¿Qué haremos pues? ¿Utilizar 10 bucles #while como los ya vistos?

Pues no. Existe una forma más corta y simple de generar los bloques de este suelo. Examinad el código de la Figura 4.

En este ejemplo declaramos una unión que emplea dos sentencias #while para crear los 200 bloques del suelo. El primer bucle es el más externo y controla el número de bloques de profundidad que han de crearse en el eje Z. (El cuerpo de sentencias de este bucle son todas las que se hallan comprendidas entre su #while y el #end más externo). Veamos cómo funciona esto: el parser de «POV» comienza a procesar linealmente una sentencia tras otra. Al llegar a la sentencia object se crea el primer objeto (como la unión está "declarada", sólo se crea el modelo en memoria) en las coordenadas determinadas por nx y nz. Luego se incrementa nx y, al llegar al #end, se retorna al último #while leído.

figura 4: "ejemplo de bucle

#while anidado"

```
#declare nz=0
#declare suelo=union{
    #while(nz<10)
        #declare nx=0
        #while(nx<20)
            object{Bsuelo200_200d1
translate<nx*200, 0, nz*200>}
            #declare nx=nx+1
        #end
 #declare nz=nz+1
    #end
}
```

Hay que comprender que las sentencias #while y #end se emparejan como los paréntesis de una fórmula. El #end en el que nos estamos fijando pertenecerá siempre al último #while "libre" que se se haya escrito antes del mencionado #end. Tened la costumbre de utilizar la tabulación para visualizar mejor las correspondencias #while-#end.

figura 5: "formatos de #if y #else"

```
//primer caso
#if (condición)
        sentencias que se compilan si la condición se cumple
#end  //fin del #if

//segundo caso
#if (condición)
        sentencias que se compilan si la condición se cumple
#else
        sentencias que se compilan si la condición no se cumple
#end  //fin del #if
```

En definitiva, el #end hallado hará que el parser retorne a considerar nuevamente la condición del segundo #while del ejemplo (al que llamamos el "más interno"). Como nx sigue siendo menor que 20 (ahora vale 1), se creará un nuevo objeto. Este proceso se repetirá hasta completar 20 pasadas en cuyo momento, al haber llegado nx a 20, la condición dejará de cumplirse y el compilador pasará a procesar la sentencia que sigue al #end. Nz pasará, entonces, a valer 1 y el siguiente #end hará que el parser retorne al #while que corresponde a dicho #end (y que en este caso es el primero y el más externo). Como nz vale 1 volverán a procesarse todas las sentencias comprendidas dentro del bucle externo y nx volverá a tomar el valor 0, con lo que todo el bucle interno volverá a ejecutarse.

Resumiendo, por cada pasada del bucle más externo, el que le sigue se ejecutará tantas veces como determine su condición (en este caso 20 veces). En cada nueva pasada del bucle externo, nz se incrementa en 1 por lo que los 20 objetos que se crean dentro del bucle más interno se sitúan 200 unidades más lejos en el eje Z que los 20 objetos de la pasada anterior; echad un vistazo a la fórmula.

Nota: en los ficheros que definen los objetos del vormiguero, 100 unidades equivalen a un metro.

En este ejemplo hemos empleado un bucle doble pero el nivel de anidación podría ser mayor. Podríamos emplear un tercer bucle dentro del segundo para que los objetos se creasen en los tres ejes. En realidad, en «POV» será muy raro que haya que emplear bucles de profundidad superior a 3 pero, según el manual, podemos llegar a escribir un superbucle anidado con 200 #while de profundidad.

#if y #else

A continuación veremos cómo funcionan las construcciones #if y #else. Los formatos más usuales de estas sentencias los podéis ver en la Figura 5.

figura 6: "ejemplos de uso de #if y #else"

```
//ejemplo A
#declare yuyu=10

#if (yuyu=452*75)
        object{bloque1}
#end
//ejemplo B
#if (yuyu=10)
        object{bloque2}
#else
        object{bloque3}
#end
object{bloque4}
//ejemplo C
#declare reyuyu=yuyu+10
#if (yuyu=10)
        #if (reyuyu=20)
        object{bloque5}
        #end
#else
        object{bloque6}
#end
```

El uso de #if es aún más simple que el de #while y el caso más frecuente es el del primer ejemplo. Todas las sentencias que quedan entre el #if y su #end forman parte del cuerpo de la sentencia condicional y serán compiladas si la condición del #if se cumple. (Estas sentencias pueden leerse como "si se cumple la condición tal, entonces compilar las sentencias del #if). En caso contrario, el parser las ignorará y pasará a compilar las que haya después del #end.

En cuanto al formato del segundo caso, funciona exactamente igual que el caso anterior salvo por el hecho de que las sentencias que siguen al #else se compilarán sólo si la condición no es cierta. En ambos casos, tanto si se compilan las sentencias atadas al #if

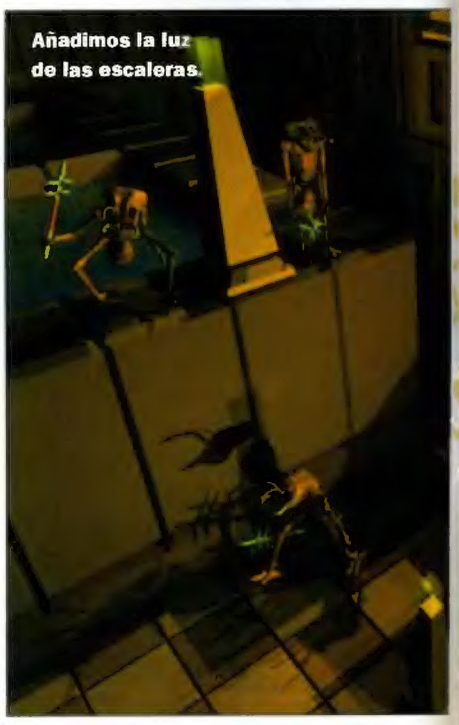

Añadimos la luz de las escaleras.

figura 7: "símbolos utilizados en las condiciones de #if y #while"

(A=B)	La condición se cumple si A es igual a B
(A!=B)	"" "" si A es diferente a B
(A>B)	"" "" si A es mayor que B
(A>=B)	"" "" si A es mayor o igual que B
(A<B)	"" "" si A es menor que B
(A<=B)	"" "" si A es menor o igual que B

(condición cumplida) como las atadas al #else (condición incumplida), el parser de «POV» continuará después con las sentencias que siguen al #end.

La sentencia #else está asociada siempre a un #if y el #end cierra la construcción definida por ambas sentencias o por un #if solitario. Ahora echad un vistazo a los ejemplos de código de la Figura 6.

En el ejemplo A, como la variable yuyu llega al #if valiendo 10, la condición no se cumple y el objeto bloque1 no se crea. En el ejemplo B se crean los objetos bloque2 y bloque4 y en el ejemplo C se crea el objeto bloque 5. Fijaos en la estructura del ejemplo C. Al igual que sucedía con #while, podemos encadenar sentencias #if y crear estructuras lógicas bastante complejas.

Recordad que –de nuevo– las sentencias #end van ligadas a los #ifs siguiendo la regla de los paréntesis (a cada paréntesis de una orientación corresponde otro de orientación distinta e idéntica profundidad). Aunque el funcionamiento de #if es muy simple, es muy fácil confundirse si tenemos muchos #ifs y #while encadenados. Intentad, pues, utilizar la tabulación para no despistaros y, si es preciso, incluid comentarios que indiquen a que #while o #if pertenece cada #end. Es importante tener en cuenta que debe haber siempre un #end para cerrar los bloques #while o #if: Si nos despistamos y ponemos un #end de más o de menos habrá problemas; tal y como sucede si nos equivocamos con el número de paréntesis de una operación.

Condiciones

Tanto #if como #while emplean símbolos para calcular si las condiciones se cumplen o no. Intentad memorizar los símbolos de la Figura 7

La verdad es que «POV» calcula si la condición se cumple o no tal y como lo hace el lenguaje C: es decir, se calcula la expresión incluida dentro de los paréntesis y si obtenemos un resultado de 0 se entiende que la condición es falsa y no debe cumplirse. Cualquier valor diferente a 0 será interpretado como cierto pero hay que recordar que los valores extremadamente pequeños (de 1e-10 o menos) pueden ser considerados por «POV» como 0. Por último, la condición a cumplir puede ser bastante más compleja de lo que se indica en la Figura 7 y así podemos escribir condiciones tales como las que pueden verse en la Figura 8.

figura 8: "ejemplos de condiciones válidas para #if y #while"

(A=B & C>B)	La condición se cumple si A es igual a B y C es mayor que B
((A=B & B<=10) | C>A)	"" si A es igual a B y B es menor o igual que 10 o bien si C es menor que A

El símbolo & debe leerse como "y" y sirve para forzar el cumplimiento de dos o más condiciones. En el segundo ejemplo hemos utilizado el símbolo | que debe leerse como "o" y que permite que las sentencias encadenadas se compilen si se cumple una condición o bien otra. (Notad que en el segundo ejemplo hay dos condiciones, no tres). Realizad vuestros propios experimentos con las condiciones de #if y #while y recordad que éstas funcionan de manera prácticamente idéntica a la de sus sentencias hermanas de C y C++.

Funciona la radiosidad de POV

En el número anterior ya comentamos que los algoritmos de radiosidad son, por ahora, los únicos que pueden simular aceptablemente la luz difusa, y que «POV» incluye una opción de radiosidad. Pues bien, en este número vamos a estudiar la radiosidad que ofrece «POV».

Antes de comenzar a estudiar las opciones que incluye la radiosidad de «POV», vamos a recordar las diferencias que presentan los algoritmos de radiosidad con respecto a los algoritmos scanline tradicionales o los de trazado de rayos.

Modelos de iluminación y algoritmos de sombreado

Llamaremos algoritmo de sombreado a cualquier método por el que se dé color a las superficies de los objetos existentes en los universos 3D generados en juegos, simulaciones o en programas de generación de imagen sintética como «3D Studio» o «POV». Existen muchos algoritmos de sombreado y cada uno presenta distintas ventajas e inconvenientes. Todos ellos se apoyan en diferentes modelos luminosos que no son sino simplificaciones que intentan simular el funcionamiento de la luz del universo real con un costo de tiempo razonable. Algunos de ellos fueron ideados pensando en limitaciones de cálculo que los ordenadores actuales han dejado muy atrás y que ya apenas se utilizan (¿quién recuerda el método de intensidad constante ò flat?). Otros, en cambio, requieren aún tanto cálculo que apenas son prácticos. Pero no vamos a ponernos ahora a ex-

La escena de ejemplo de Dan Farmer generada con radiosidad.

La misma escena con trazado de rayos.

plicar cómo funciona el sombreado Phong o en qué se basan los modelos luminosos de Whitted. Hall y otros investigadores. En lugar de esto vamos a intentar establecer una sencilla clasificación para estos algoritmos.

Por un lado, tenemos los llamados algoritmos scanline o de línea de rastreo, luego están los de trazado de rayos y por último se encuentran los métodos que emplean radiosidad.

En los algoritmos scanline el método empleado consiste en hacer un barrido de la imagen de arriba a abajo. Previamente podemos haber empleado cualquier algoritmo para determinar las superficies visibles pero no es infrecuente que en estos casos se utilice también el algoritmo de eliminación de superficies ocultas que lleva también el nombre de línea de rastreo. Ciñéndonos a lo básico, cada pixel de las superficies visibles será coloreado atendiendo al ángulo que presenta con respecto a las fuentes de luz que lo iluminan. También suelen tenerse en cuenta otros factores como los colores de los objetos y de las fuentes, la atenuación de la luz con la distancia, etc. Por último, conviene reseñar que casi todos los algoritmos de este tipo no aplican el modelo luminoso completo (es decir, todas las fórmulas) en cada punto, sino que suelen realizarse interpolaciones con el color de los puntos (Goraud) o con las normales (Phong) para ahorrar tiempo.

En el trazado de rayos, en cambio, el sistema es muy diferente y mucho más complicado. Para generar una escena el programa lanzará rayos desde la teórica posición del observador. Se trazará, al menos, un rayo para cada pixel de la escena y cada uno será seguido para comprobar si choca con la superficie de algún objeto. Esto implica ya muchos cálculos, pues no es tan fácil determinar cuándo un rayo ha chocado o no con superficies de forma compleja. Por ello suelen emplearse métodos como el de las cajas bounding para minimizar los cálculos. En este método cada objeto es encerrado dentro de una caja invisible que tiene sus mismas dimensiones máximas. Esto se hace así porque a menudo es mucho más rápido comparar choques con cajas que con los objetos que éstas contienen. Además, si el número de objetos es muy grande, entonces puede establecerse un nivel de jerarquía mayor para ahorrar tiempo.

En el caso de nuestras escenas de batallas, por ejemplo, «POV» podría estar empleando una caja para cada guerrero. El rayo comprobaría el posible choque con respecto a cada cubo-guerrero y luego, cuando se hubiese determinado el choque con el cubo contenedor de un soldado en particular, entonces se harían comprobaciones de posición con las cajas contenedoras "de menor jerarquía" de los objetos componentes (manos, antebrazos, etc.) de ese soldado en particular. Sólo cuando se hubiese llegado a determinar el choque del rayo con una de estas cajas "de menor jerarquía" procedería el programa a realizar los lentos cálculos rayo-superficie con el objeto contenido dentro de la caja en cuestión; esto ahorra numerosos cálculos.

Una vez que el rayo del observador ha chocado con una superficie se lanzan los llamados "rayos de sombra" desde el punto del choque. Se enviará un rayo de sombra por cada fuente de luz que exista en la escena y cada rayo se dirigirá en línea recta hacia una fuente concreta para comprobar si el camino hacia dicha fuente está despejado o no. Aquí de nuevo se gana tiempo con el sistema de las cajas contenedoras. Con ello se comprobará qué fuentes iluminan al punto de choque en cuestión y las intensidades se sumarán para determinar el color del punto. La cosa puede complicarse mu-

cho si estamos empleando áreas de luz. La cosa se complica aún más por el hecho de que el trazado de rayos también trata la reflexión y la refracción de la luz. Así, cuando un rayo primario –uno de los lanzados originalmente– choca con una superficie, se lanza un nuevo rayo, al que denominaremos reflejado. Este nuevo rayo parte del punto de choque siguiendo una trayectoria especular; lo que significa que abandonará la superficie impactada con el mismo ángulo con que el rayo primario ha llegado a ella. De esta manera, el programa seguirá al nuevo rayo para comprobar si choca con una nueva superficie. Si se da este caso se lanzarán rayos de sombra desde el nuevo punto de choque para comprobar si las fuentes lo iluminan y la intensidad con que lo hacen. El tono de color de este punto se sumará al del primer punto de choque del que partió el rayo reflejado. ¡Pero esto no es todo! También se lanzará un nuevo rayo reflejado desde el segundo punto de choque y esto continuará hasta llegar al valor máximo de recursión que hayamos estipulado para el programa. En «POV», por defecto, es de 5, aunque podemos cambiar este valor con la variable max_trace_level, dentro de la sentencia global_settings.

Pero todo esto no es más que una simplificación. Ni siquiera hemos hablado de los rayos transmitidos que se lanzan cuando las superficies impactadas son total o parcialmente transparentes, pero lo dicho es suficiente para comprender de dónde toma su nombre el trazado de rayos y para saber por qué se consiguen imágenes más realistas con el ray-tracing que con los métodos scanline. Es cierto que también pueden conseguirse reflexiones con estos métodos, pero normalmente hay que indicarlo al programa y al hacerlo estaremos perdiendo parte de la única ventaja que tiene el rastreo de línea sobre el trazado de rayos: el menor tiempo de cálculo.

¿Y qué es la radiosidad? Pues es un modelo de iluminación que se basa en la termodinámica. Los métodos basados en la radiosidad estudian la distribución de energía en las superficies de ambientes cerrados. En estos métodos todas las superficies pueden ser fuentes emisoras de luz u objetos reflectores o ambas cosas. De esta manera, calculando la cantidad de energía radiante que llega a cada área procedente de fuentes de luz o de superficies reflejantes, se puede representar la iluminación difusa en las escenas. Como recordaréis, este tipo de iluminación no se calcula con los métodos anteriores y, en su lugar, para evitar que las áreas no iluminadas directamente queden a oscuras, suele emplearse un burdo truco llamado iluminación ambiental constante.

Los métodos de radiosidad suelen conseguir resultados aún más realistas que los logrados con el trazado de rayos pero, como contrapartida, el tiempo de cálculo se dispara aún más. La radiosidad es un área que todavía se está investigando, y de hecho los programas de radiosidad pura no son considerados demasiado prácticos por lo que, casi siempre, la radiosidad suele combinarse con el trazado de rayos. Esto es lo que sucede en «POV», donde la radiosidad es un precálculo que se efectúa sobre la escena antes de lanzar rayos de la forma antes descrita.

Cuando el valor de count es demasiado bajo la radiosidad empieza a perder calidad.

La radiosidad de POV

Por defecto, la radiosidad de «POV» está desconectada. Esto se debe a que el algoritmo empleado es experimental y quizá sea cambiado en el futuro (aunque parece que todo sigue igual en la nueva versión 3.1). Naturalmente, esto implica que los archivos de escena que se rendericen ahora con radiosidad

pueden producir imágenes diferentes en futuras versiones de «POV».

El algoritmo de radiosidad implementado en «POV» está basado en el método de Greg Ward cuyos detalles exactos de funcionamiento todavía nos son desconocidos. Para hacer vuestros experimentos lo mejor será hacer pruebas con el fichero rad2.pov que hallaréis en el directorio \pov3demo\radios de vuestra instalación de «POV». Comenzad renderizando este fichero ordenando una resolución de 320*200 pixels y examinad el resultado. La escena generada es una pequeña habitación con paredes de color rojo y azul, techo blanco y suelo a cuadros verdes y azules. Los únicos objetos son una esfera blanca situada cerca de la pared roja y una especie de ladrillo amarillo colocado cerca de la pared azul. Pero, ¿dónde están los efectos de la radiosidad? La verdad es que todavía no los hay.

Para activar la opción de radiosidad hemos de incorporar el comando +Qr a la línea de órdenes con que invocamos al programa. Antes, sin embargo, editad el archivo rad2.pov. Veréis al principio una serie de sentencias desactivadas con el símbolo "//" del comentario y que comienzan así: "//#declare Rad_Quality= Radiosity_". Más abajo está la línea:

//#include "rad_def.inc"

Para realizar nuestro primer experimento quitad la marca "//" de comentario de esta última línea y de la que asigna a Rad_Quality el valor Radiosity_Normal. Después, invocad a «POV» con el comando +Qr y no olvidéis dar otro nombre al fichero de salida para poder comparar la imagen resultante con la anterior. Una vez que la escena haya sido renderizada, comparadla con la anterior. Al principio ambas imágenes os parecerán casi iguales pero los cambios, aunque sutiles, se harán evidentes inmediatamente. Fijaos, por ejemplo, en la esfera blanca: en la escena generada con radiosidad el lado pegado a la pared roja tiene un leve tono rojizo mientras que el lado opuesto, que tiene enfrente una pared azul, exhibe un tono azulado. Asimismo, en esta escena el ladrillo amarillo es mucho más oscuro en la cara que queda frente a la pared azul. Además, hay también otros cambios más difíciles de percibir: el techo y la pared blanca de la habitación han tomado leves matices rojos y azules a causa de las otras dos paredes.

Ejemplo usando el valor de 800 para count.

Todos estos cambios se deben a las reflexiones difusas que «POV» sólo calcula cuándo se activa la opción de radiosidad. Como recordaréis, en el número anterior, llamábamos luz difusa o ambiental a la que se producía como consecuencia de las múltiples reflexiones de los rayos de luz con los objetos. En el mundo real, los rayos de luz chocan contra los objetos y rebotan una y otra vez. Este efecto es simulado en los métodos de radiosidad lanzando rayos entre las superficies para determinar el intercambio de energía radiante en el espectro visible. Esto significa dos cosas: la primera que habrá iluminación en zonas que no resultarían iluminadas con el simple trazado de rayos, y la segunda, que el color de la luz que radian las superficies se sumará a la iluminación directa proporcionada por las fuentes de luz. Es esta última consecuencia lo que hace que la esfera reciba iluminación indirecta roja y azul de las paredes.

Algunas pruebas de radiosidad

Pero hagamos más pruebas. En «POV» podemos controlar ciertos aspectos de la radiosidad con una serie de sentencias que deben ir incluidas entre las llaves de la instrucción "Radiosity". A su vez, esta sentencia figura entre las que pueden escribirse dentro de la instrucción global_settings, utilizada para ajustar ciertos parámetros en la generación de la imagen, como el factor gamma, el nivel máximo de recursión de los rayos reflejados a lanzar, etc. Así, el formato general de esta sentencia lo podéis ver en la Figura 1 (página siguiente).

Si queréis ver algunas pruebas hechas con valores predeterminados edi-

Figura1: "Formato de la sentencia radiosity utilizada para ajustar la opción de radiosidad"

```
global_settings {
  ...
//Nota: no es preciso colocar todas las sentencias
  radiosity {
  brightness FLOAT
  count INTEGER
  distance_maximum FLOAT
  error_bound FLOAT
  gray_threshold FLOAT
  low_error_factor FLOAT
  minimum_reuse FLOAT
  nearest_count INTEGER
  recursion_limit INTEGER
```

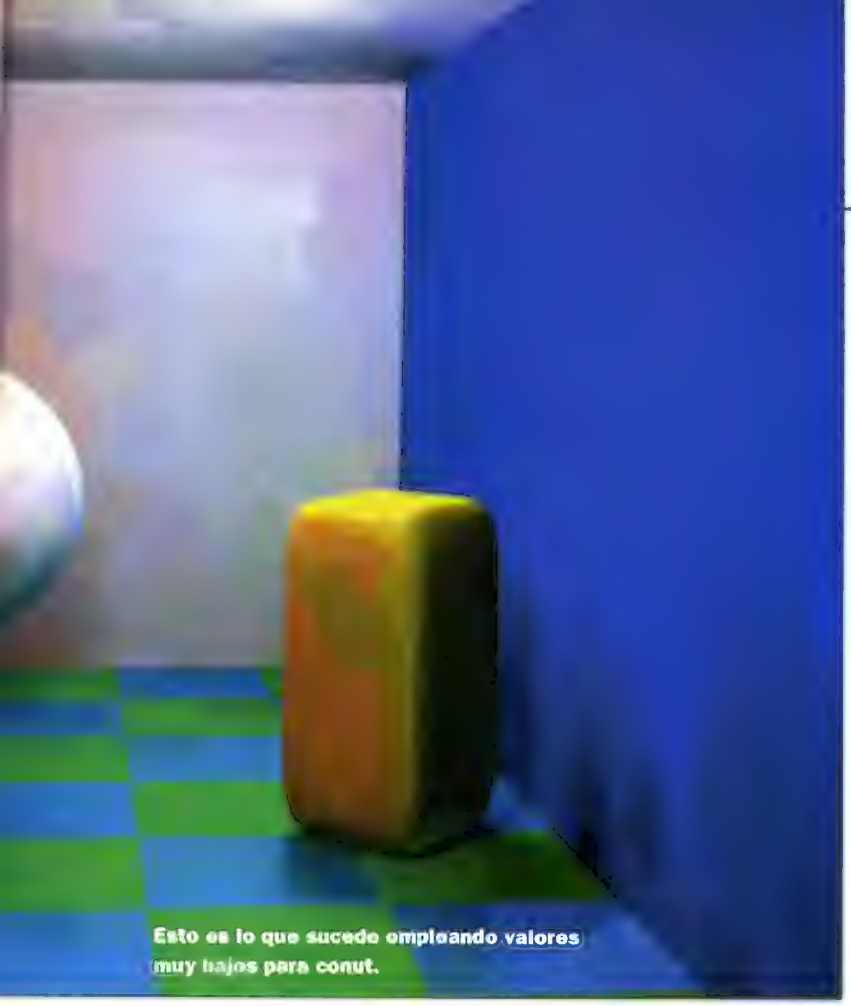
tad el fichero rad_def.inc situado en el directorio include de la instalación de «POV». Este archivo es "incluido" por rad2.pov, tiene guardados varios ejemplos para la sentencia radiosity, y ejecutará uno u otro dependiendo del valor que activemos en rad2.pov para la variable Rad_Quality. Así, si activamos (quitando los caracteres de comentario) el valor Radiosity_Normal, se emplearán valores medios para la radiosidad. Con Radiosity_Fast obtendremos un resultado más rápido pero con una calidad inferior, con Radiosity_Final obtendremos unos resultados de mucha más calidad (pero más lentos), etc.

La verdad es que, normalmente, los cambios producidos por las sentencias incluidas en el bloque radiosity son prácticamente imperceptibles, y por ello vamos a prestar atención sólo a algunas de ellas.

Editad el fichero rad3.pov, donde hemos incluido un bloque global_settings para hacer nuestros propios experimentos, y preparaos para trastear. Nuestra primera prueba consistirá en cambiar el valor que sigue a la sentencia count por el valor 10. Renderizad el archivo y comparadlo con el anterior (que se renderizó con un count de 200). Como veréis, el render tardará mucho menos tiempo pero los resultados serán bastante inaceptables. El tono general de distribución de luz será tan correcto como en la prueba anterior, pero los colores estarán dispuestos a brochazos. Esto se debe a que el algoritmo de radiosidad lanza rayos desde cada punto con ángulos determinados para establecer la energía que recibe ese punto.

Con gray_threshold a 0.10, la radiosidad gana color.

Si el valor de count es muy bajo o si hay una gran diferencia de tamaño entre los objetos de la escena, entonces ésta no se calculará correctamente. Un valor de 200 será normalmente el adecuado para count pero si apreciáis un efecto parecido al de esta última prueba, probablemente tendréis que elevar este valor. También debéis procurar que las desigualdades de tamaño entre los modelos de la escena no sean excesivos. No es conveniente, por ejemplo, enfocar una manzana sobre el suelo de un patio de 40 metros cuadrados. ¿Por qué? Porque cuanto mayores sean las diferencias de tamaño entre los objetos de la escena, mayores serán los ángulos de los rayos "de radiosidad" que habrá que lanzar y mayor será la probabilidad



Esto es lo que sucede empleando valores muy bajos para conut.

de que la escena tenga fallos, al haber zonas que no reciban estos rayos.

En teoría podemos corregir el problema subiendo el valor de count, pero en la práctica el aumento de tiempo puede convertir esta opción en algo poco práctico. (Nota: hace ya bastante tiempo tuvimos el mismo problema con «Raysmith», un programa de radiosidad creado por un autor español. En general, la falta de tiempo nos obligó a generar escenas con pocos rayos y por ello las imágenes tuvieron menos calidad de la que deberían haber tenido). Aquí hay que decir que «POV» dispone de una sentencia que supuestamente puede arreglar este problema sin necesidad de aumentar el valor de count. Se trata de distance_maximun. El valor que sigue a este parámetro está relacionado con el tamaño de la zona enfocada y de la distancia de la cámara a la misma. Sin embargo, reconocemos que aún no sabemos emplearla adecuadamente, por lo que tuvimos que dejar este valor a 0 y limitarsnos a trastear con el valor de count.

Otra sentencia que afecta a la calidad de la escena es recursion_limit, que controla la recursión de los rayos, pero lo mejor será que dejéis su valor a 1 a no ser que dispongáis de un Pentium VI a 3000 MHz.

Por último, es interesante jugar con el valor de gray_threshold. En el universo real, los millones de reflexiones de rayos que se producen entre superficies de distinto color hacen que se produzca una mezcla de color por la cual normalmente la luz ambiental tiende a tomar un tono gris. «POV» lo simula con el valor de este parámetro.

Probad, por ejemplo, a bajar el valor de gray_threshold a 0.10 en nuestra escena experimental. Como veréis, los colores resultantes de la iluminación difusa serán mucho más vivos (podréis apreciarlo si os fijáis en los tonos rojos y azules de la esfera). Si, en cambio, probáis con un valor de 0.90 o mayor para este parámetro, entonces los tonos serán casi grises.

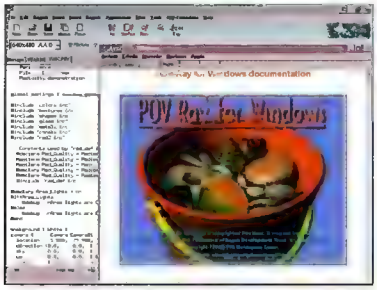
En definitiva, un valor muy bajo de gray_threshold hará que la escena parezca casi de dibujos animados, mientras que un valor próximo a 1 provocará que el color de la luz ambiental desaparezca. Es recomendable utilizar valores de 0.35 a 0.65, según los casos.

Sobre los experimentos

Todos los experimentos iniciales se realizaron con el archivo de prueba rad2.pov. Sin embargo, también se lanzaron algunas pruebas con los escenarios de interiores que se crearon para el presente número. La idea inicial para estos escenarios era crear algo basado en la arquitectura del egipto de los faraones, pero esta idea acabó en la papelera cuando el autor vió el magnífico mapa lkdm4 para «Quake 2» de Iikka Keranen. Este mapa estaba lleno de pasillos y salas muy sencillas construidas con muy pocos elementos que se repetían en suelos, techos y paredes. Sin embargo, a pesar de esta sencillez, el mapa resultaba precioso y sus estructuras resultaban, además, fáciles de reproducir con las sentencias de programación. Por otra parte, dichas estructuras eran también ideales para hacer pruebas de radiosidad.

Nota: desgraciadamente, el segundo ordenador con el que esperábamos hacer estas pruebas, un Pentium II a 266 MHz, se estropeó y sólo pudimos contar con nuestro viejo Pentium a 150 MHz. Por esta razón, no hemos podido dejar los escenarios tan bien como hubiésemos deseado.

POVRAY 3.1
¿Un lenguaje de programación?



Aunque estaba previsto comenzar este mes con el especial de animación, hemos decidido retrasarlo por dos razones: el espacio que ha ocupado el artículo anterior y... por la salida de la beta de la próxima versión de «POV». Efectivamente, «POV 3.1» ya está disponible en http://www.povray.org y trae algunas novedades que harán furor entre nuestros pov-adeptos.

En más de una ocasión hemos comentado que la estructura del lenguaje escénico de «POV» recuerda a la de un lenguaje de programación y, en particular, a la de C. Pues bien, parece que esta semejanza ha aumentado en la nueva versión 3.1 (y presumiblemente seguirá haciéndolo en el futuro). La nueva beta (que estará vigente hasta septiembre) incorpora arrays, macros, funciones de apertura, lectura y grabación de ficheros de texto y variables locales. Incluso el inevitable punto y coma del final de las sentencias de C se usará también en muchas de las sentencias del lenguaje de «POV». Sin duda, cualquier amante de «POV» que domine C comprenderá lo que significan estos cambios.

Aparte de esto también hay algunas novedades en el apartado de las texturas, pero lo comentado arriba va a suponer cambios revolucionarios entre los amantes de la infografía y la programación. ¿Por qué? Bueno, para empezar las nuevas sentencias de lectura y grabación nos van a ser útiles para un montón de cosas. Según el manual provisional de «POV», las nuevas sentencias #fopen, #read, #write y #fclose fueron creadas como una ayuda para el apartado de animación (para pasar información al frame siguiente y cosas así) pero lo cierto es que podrían ser empleadas para muchas cosas más. Se podría, por ejemplo, leer archivos de datos posicionales para fijar objetos como árboles o personajes en una geografía accidentada (cuyos valores de alturas se hubiesen calculado previamente).

En cuanto a los arrays, podrían ser empleados para almacenar estructuras de datos con, por ejemplo, valores con las dimensiones de cajas contenedoras para los objetos. De esta manera, leyendo y procesando los datos de estos arrays, podríamos crear escenas con miles de objetos complejos y asegurarnos de que jamás habría superposición espacial. Además, si estos arrays tuviesen datos relativos al terreno, quizá podríamos crear animaciones de personajes cuyos pasos se adaptasen al terreno.

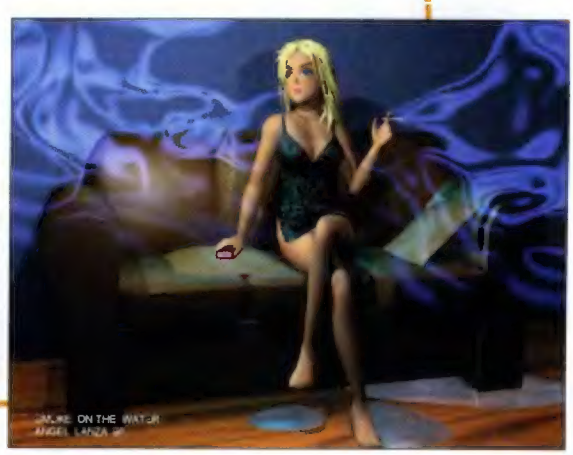
Otro avance es la implementación de macros. Con ellas parece ser que por fin tendremos algo muy parecido a las funciones de C sin necesidad de sacar código a otros ficheros, como hasta ahora. Simplemente escribiremos las macros y luego, al invocarlas, éstas se expandirán con los cambios que se indiquen en sus parámetros. Sin duda, esta posibilidad será una de las que más alegrará la vida a los escritores de pov-ipas. En fin, hay aún más cambios, pero nos hemos quedado sin espacio. Hablaremos de todo esto en próximos números.

Nota importante. Podéis remitirnos vuestros trabajos o consultas, bien por carta a la dirección que figura en el sumario de Pcmanía, o vía e-mail a rendermania.pcmania@hobbypress.es

El regreso del Mech

En el foro de este mes, un buen porcentaje de los modelos enviados son mechs muy variados. Además, nuestra amiga Pamela volverá a honrarnos con su deliciosa presencia y también contaremos con otros interesantes trabajos, entre los que hay bicicletas, templos griegos, cuadros abstractos e incluso un nuevo pov-ipas.

De nuevo está con nosotros **Jesús Ángel Lanza Salcines** que nos envía una nueva página HTML donde volveremos a disfrutar de la compañía de la ya famosa Pamela. ¿Qué más se podría decir de esta maravillosa chica virtual de 7266 vértices? ¡No os perdáis estas fantásticas pam-escenas!



El Foro del Lector

Nuestro amigo **Josh** es otro recién llegado al mundillo de la imagen sintética. Sus primeras imágenes han sido modeladas con «Spatch» y renderizadas desde «Imagine». Según afirma en su carta, Josh no ha podido tirar los renders desde «POV» porque no tiene este programa y también carece de conexión a Internet. ¿Que por qué no hemos publicado «POV»? Bueno, PCmanía ha publicado este raytracer varias veces y tiene intención de volver a hacerlo pero lo cierto es que el Povteam (sus creadores) aún no han contestado nuestra última carta solicitando su permiso para ello. Esperamos poder publicarlo muy pronto. En cuanto a tus otras preguntas:

1) Las texturas procedurales de «POV» son, en mi opinión, mejores que las de «Imagine» porque el primero posee más sentencias para crear texturas de este tipo que opciones tiene «Imagine» para hacer lo propio. Además, la calidad del render de «POV» es superior.

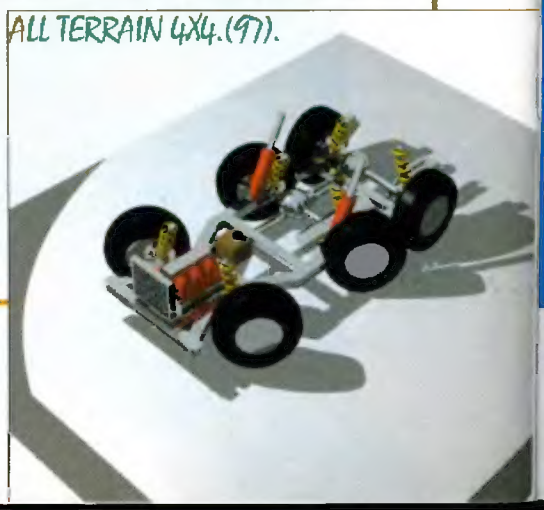
2) En el artículo de este mes dedicado a la radiosidad se resumen las diferencias entre los métodos scanline, y los de trazado de rayos y de radiosidad.

3) ¿Los fallos más grandes de tu trabajo? Bueno, mas que de fallos yo hablaría del camino que todavía te queda por recorrer. Tu primer paso debería ser coger más soltura en el modelado, después te sería conveniente aprender a ambientar mejor las escenas: la iluminación, las texturas, la atmósfera, etc.

Juan Luis Valle nos envía unas escenas hechas con «3D Studio 4» entre las que se incluye un mech de su invención. Juan Luis adjunta su dirección para intercambiar modelos y texturas con otros rendermaníacos y amenaza con enviar sus próximas escenas a una resolución de 1600*1200 (por favor, no excedas los 800*600 pixels). En cuanto al envío de los modelos, creo que si hubieses empleado un número cuatro veces menor de polígonos, el mech no hubiese perdido detalle; no te apures, el emplear un exceso de caras es una manía muy corriente. Hasta la próxima.

Javier Pla Saavedra nos envía algunas escenas realizadas con «Corel Dream 3D» y pregunta si existen conversores para pasar los modelos de este programa a otros formatos más convencionales con el fin de participar en alguna de las próximas batallas.

No sé nada de este programa ni de posibles conversores para él, aunque apostaría a que debe tener una opción de exportación para el formato DXF y eso es todo cuanto necesitas para que podamos emplear tus modelos (aunque a veces los modelos grabados en este formato dan muchos problemas). Respecto a lo que comentas, existe una versión recortada de «3D Studio», aunque PCmanía por ahora no va a publicarla. Y te recomiendo el libro «3D Studio 4» de Anaya.



Benyi Albares, otro viejo conocido nuestro, vuelve a estas páginas con tres magníficas escenas creadas con «3D Studio», «Max», «Metaballs 2» y «Bryce 2» (al que no conozco). Tu chica me ha gustado mucho (aunque hubiese preferido verla de cuerpo entero) y me ha impresionado el extraordinario realismo del agua de metalsIII (¿creada con «Bryce»?). En cuanto al envío de obras, preferimos que las enviéis en formato JPG y a 800*600 pixels, aunque no es obligatorio.

Por otro lado, Benyi ofrece a todos los rendermaníacos la posibilidad de publicar sus obras en Internet, en una exposición gratuita (leed su carta en el foro). Y por último, por lo que respecta al desafío entre mechs del que hablas, podéis utilizar las reglas explicadas en la última página de Rendermanía 18 –al menos hasta que alguien envíe sugerencias para mejorar dichas reglas–. Por supuesto espero esas animaciones tuyas , pero recuerda que preferimos que las animaciones –ya comprimidas– no excedan los 5 Megas.

José Mª González Romero, alias Narcosis, nos envía una magnífica bicicleta modelada y renderizada con «POV» y algunas preguntillas:

1) Es cierto que algunos de los últimos artículos sobre «POV» son algo duros de digerir pero ello se debe más a la necesidad de resumir que a la complejidad de los mismos. Ten en cuenta que de «POV» podría escribirse tranquilamente un libro de 600 páginas sin necesidad de incluir paja dentro del mismo.

2) Es muy raro que yo utilice más de dos fuentes de luz para una escena y realmente esto es lo que mejor suele quedar si se consigue un buen contraste entre las zonas oscuras y las iluminadas. Si sitúas un exceso de luces lo único que conseguirás será que toda la escena te quede empastada y sin tonos . ¡Pero la colocación de estas luces no es nada difícil! En general, lo más simple es tomar nota de la situación de la cámara y dar las mismas coordenadas a una de las fuentes, sumando una pequeña variación posicional.

3) La apariencia de las texturas depende no sólo de las propiedades asignadas a las mismas sino también del color y número de las fuentes de luz. ¡No pongas demasiadas, en tu bici has puesto 6 cuando podrían haber bastado 2!

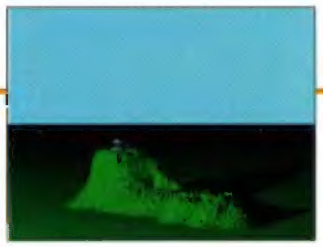
4) En cuanto a la diferencia entre las imágenes impresas y las equivalentes del CD. La verdad es que a veces algunas escenas llegan tan oscuras que no hay más remedio que subirles el contraste o iluminarlas un poco para que resulten mínimamente visibles una vez impresas. Después del proceso de impresión, una foto no sale nunca igual que el fichero gráfico del que ha partido.

El mech enviado por **Luis Trigo Prunera** y creado con «3D Studio» tiene probablemente el diseño más simpático que hemos visto aquí (la escena donde su criaturilla persigue al marine espacial de Roberto Celorrio es muy divertida). Luis incluye, además, una carta explicando los detalles de modelado y también al mismo modelo para que sea libremente utilizado por cualquiera. Gracias Luis. lo utilizaremos en la próxima batalla.

El Foro del Lector

Jorge Reguillo de Luis también se apunta a la creación de mechs y nos envía una reproducción bastante exacta del famoso dreadnought eldar de Warhammer 40000 y de la que incluye las mallas en formato 3ds. ¿Una crítica constructiva? Bueno, no tengo la miniatura en la que está basada el mech, aunque sí fotos de White Dwarf, pero yo diría que tu modelo está muy bien, aunque quizá te faltan algunas piezas en los hombros y algo de retoque en las manos. Claro que con lo difícil y agotador que es modelar manos convincentes no seré yo el que te eche esto en cara. Las de mi último modelo son aún peores.

Javier Pérez Martín nos remite dos estupendas animaciones y un cañón de barco del que nos pide una crítica feroz. Bueno, tal como dices, lo peor es la textura del agua. La boca del cañón resulta también algo extraña aunque el artefacto en sí no está mal. En cuanto a la versión de «3D Studio», estamos en ello.

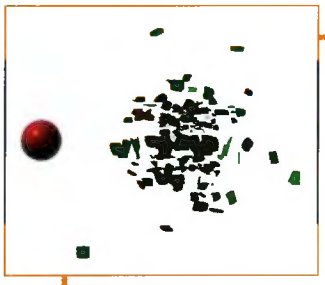


Ramón Rodríguez es otra víctima del virus infográfico. Domina otros programas como «3D Studio» y «Caligary», pero nos envía algunas escenas construidas íntegramente con «POV» e incluye algunas preguntas:

1) ¿Cómo se pueden crear montañas con «POV»? Pues, aparte del método que has empleado, podías haber generado una TGA que sirviese como archivo de entrada para la sentencia heightfield, en lugar de dibujar la TGA a mano. Esto puedes hacerlo enfocando e iluminando perpendicularmente un plano al que hayas puesto una textura adecuada. Otra posibilidad es usar alguna de las utilidades que existen para crear montañas como Terrain Maker o Hf-lab (Rendermanía 8).

2) En cuanto a la animación con «POV», este mes hemos comenzado a tratar este tema.

3) Y por último, en cuanto a la versión limitada de «3D Studio», aún no la hemos publicado, pero nunca se sabe...



De extremadamente interesante puede calificarse al POV-ipas T.N.T. versión 2.0 remitido por **José Francisco García**. Este trabajo es un archivo de tipo include que puede ser utilizado por los povmaníacos para explosionar sus objetos en tantos pedazos como sea preciso. José Francisco incluye, además, un completo fichero de ayuda para socorrer al sufrido usuario y unas prometedoras escenas de ejemplo. Bueno, lo cierto es que ya dispongo de varios ficheros de este tipo, así que es posible que próximamente se haga un especial sobre IPAS explosivos en el que, por supuesto, participará esta utilidad.